2
Trattare l'intrattabile

In questo capitolo discuteremo di come affrontare problemi per i quali non abbiamo algoritmi efficienti. Per tali problemi dobbiamo necessariamente usare un approccio non efficiente. Tuttavia, in molti casi, possiamo utilizzare degli accorgimenti che limitano l'inefficienza.

2.1 Introduzione

Supponiamo che vi venga dato un problema per il quale dovete fornire un algoritmo. Dopo aver pensato un po' ed aver provato ad usare le tecniche imparate durante i corsi di algoritmi vi accorgete di non essere riusciti a cavare un ragno dal buco. E se il problema fosse di quelli difficili, cioè un problema NP-hard? Sarebbe uno di quelli per i quali nessuno è riuscito a trovare un algoritmo efficiente perché magari tale algoritmo non esiste!

Quando null'altro funziona, possiamo usare la tecnica meno creativa possibile: provare tutte le possibilità. Fare cioè una *ricerca esaustiva* su tutto lo spazio delle soluzioni. Tale approccio, noto anche come *ricerca a forza bruta*, in linea teorica risolve qualsiasi problema. Ovviamente lo fa usando molto tempo. Non è difficile fornire degli esempi in cui una tale tecnica, usando anche i computer più veloci finora costruiti, impiega anni, secoli, millenni, migliaia di millenni e più per risolvere il problema.

Consideriamo ad esempio il problema della fattorizzazione. Dato un numero intero maggiore di 1, stabilire quali sono i suoi fattori. Questo problema ha un'enorme importanza pratica in quanto la sicurezza di molti sistemi crittografici si basa sulla difficoltà di fattorizzare un numero $N = a \cdot b$ che è ottenuto come prodotto di due numeri primi a e b molto grandi (considerando le capacità computazionali dei computer attuali, si usano numeri dell'ordine di 2048 bit). Se il numero N che ci viene fornito come input è piccolo, non è difficile scoprire i suoi fattori e quindi anche a e b nel caso in cui N sia il prodotto di due primi. Ad esempio se $n = 221$ l'approccio a forza bruta, che consiste nel dividere N per tutti gli interi a partire da 2 fino a che non si trova un divisore (che può essere N stesso se N è primo), non impiegherà molto tempo per scoprire che $221 = 13 \cdot 17$. Tuttavia più grande è N e più tempo servirà a trovare i fattori di N. Si noti che la "taglia" del problema della fattorizzazione è di $n = \log N$ bit e quindi provare a dividere per tutti i valori da 2 a N costa tempo esponenziale rispetto

alla taglia dell'input e quindi basta aumentare N per arrivare abbastanza rapidamente a tempi di esecuzione di giorni, mesi, anni, secoli o millenni (e anche più).

2.2 *Esercizio di programmazione*

Scrivere un programma che esamina un numero N per stabilire se è primo oppure se è il prodotto di fattori dividendolo per tutti i numeri più piccoli di $N/2$. Nel caso N sia il prodotto di fattori si stampino i fattori. Si faccia anche stampare al programma il tempo che impiega per risolvere il problema. Si utilizzi il programma per capire in che modo varia il tempo di esecuzione al crescere della taglia dell'input. Nel riquadro è riportato un programma Java.

```
import java.util.Scanner;
import java.math.BigInteger;
import java.util.*;
public class Fattorizzazione {
  public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Inserisci un numero: ");
      BigInteger n = scanner.nextBigInteger();
      BigInteger ZERO = BigInteger.ZERO;
      BigInteger TWO = BigInteger.valueOf(2);
      if (n.compareTo(TWO)<0) {
        System.out.println(String.format("Il numero inserito %s e' minore di 2",n.toString()));
        System.exit(0);
      }
      long t1,t2;
      List divisori = new ArrayList();
      t1 = System.currentTimeMillis();
      System.out.println(String.format("tempo ms: %s",t1));
      BigInteger nhalf = n.divide(TWO);
        for (BigInteger i = TWO; i.compareTo(nhalf) < 0; i = i.add(BigInteger.ONE)) {
        BigInteger risultato[] = n.divideAndRemainder(i);
        if (risultato[1].compareTo(ZERO)==0) {
          divisori.add(i);
        }
      }
      if (divisori.size()==0) {
        System.out.println(String.format("Il numero inserito (%d) e' primo", n));
      }
      else {
        System.out.println(String.format("Il divisori di %d sono "+divisori, n));
      }
      t2 = System.currentTimeMillis();
      System.out.println(String.format("tempo ms: %s",t2));
      long delta=t2-t1;
      System.out.println("Millisecondi: "+delta);
  }
}
```

2.3 *Ricerca esaustiva intelligente*

Se proprio dobbiamo usare una ricerca esaustiva possiamo sfruttare degli accorgimenti che permettono di migliorare il tempo di esecuzione (che rimarrà comunque non polinomiale). Durante la ricerca esaustiva possiamo eliminare grossi sottoinsiemi di potenziali soluzioni in un solo colpo diminuendo così il tempo totale. Ad esempio, nell'esercizio precedente abbiamo risolto il problema tentando di dividere N per tutti i numeri più piccoli fino ad $N/2$, scartando a priori i numeri più grandi di $N/2$ in quanto essi non possono dividere N. Il vantaggio in questo caso è stato quello di dimezzare l'insieme dei potenziali divisori di N. Ovviamente, in questo caso, basta raddoppiare il valore di N per vanificare questo piccolo vantaggio. Tuttavia, anche se piccoli, i vantaggi possono essere significativi in pratica. Le tecniche conosciute con i nomi di *backtracking* e *branch and bound* permettono di ottenere dei miglioramenti. Il backtracking si basa sull'osservazione che a volte è possibile rifiutare una potenziale soluzione esaminando solo una piccola parte che però è sufficiente per capire che non siamo di fronte ad una soluzione del nostro problema. Analogamente il branch and bound applica lo stesso principio a problemi di ottimizzazione, scartando in un solo colpo tutte le soluzioni per le quali siamo in grado di dire, senza esaminarle nei dettagli, che il loro valore non può essere quello ottimale. Nel seguito faremo due esempi per illustrare queste tecniche.

2.4 *Backtracking*

2.4.1 Sat

Consideriamo il problema Sat introdotto nel capitolo precedente: data una formula booleana stabilire se esiste un assegnamento delle variabili che la rende vera. Se un tale assegnamento esiste, la formula si dice soddisfacibile. Per semplicità considereremo questo problema nella sua formulazione in forma normale congiuntiva, cioè usando formule booleane che sono costruite come AND di OR. Questa non è una restrizione in quanto una qualsiasi formula può essere espressa in forma normale congiuntiva.

Per essere concreti nella spiegazione, consideriamo un esempio:

$$\phi = (a + \neg b + c + \neg d) \cdot (a + b) \cdot (a + \neg b) \cdot (\neg a + c) \cdot (\neg a + \neg c) \tag{2.1}$$

Dato uno specifico assegnamento di valori alle variabili a, b, c e d è facile verificare se ϕ è vera o falsa. Basta applicare la formula. Ad esempio se $a = 1$, $b = 0$, $c = 1$, $d = 0$, si ha che

$$\begin{aligned} \phi &= (1 + \neg 0 + 1 + \neg 0) \cdot (1 + 0) \cdot (1 + \neg 0) \cdot (\neg 1 + 1) \cdot (\neg 1 + \neg 1) \\ &= (1 + 1 + 1 + 1) \cdot (1 + 0) \cdot (1 + 1) \cdot (0 + 1) \cdot (0 + 0) \\ &= (1) \cdot (1) \cdot (1) \cdot (1) \cdot (0) \\ &= 0 \end{aligned}$$

Per trovare un assegnamento che renda ϕ vera o per stabilire che non esiste un tale assegnamento possiamo usare una ricerca esaustiva: ci sono 16 possibili assegnamenti delle 4 variabili. Possiamo controllarli tutti per scoprire se uno di essi rende la formula

vera oppure se non esiste nessun assegnamento che rende la formula vera. Se guardassimo tale processo con un albero delle decisioni, che in questo caso possiamo a ragione chiamare albero degli assegnamenti, in cui ogni volta decidiamo se assegnare valore 1 o 0 ad una delle variabili, le 16 possibilità corrisponderebbero all'albero mostrato nella Figura 2.1.

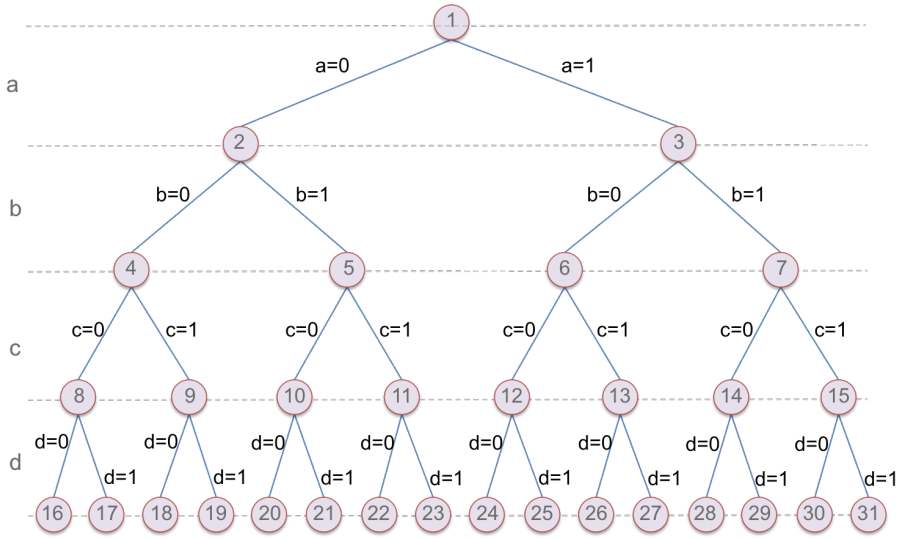

Figura 2.1: Albero degli assegnamenti

In questo esempio ci sono solo 16 foglie nell'albero degli assegnamenti. In generale ci saranno 2^n foglie, dove n è il numero di variabili usate nella formula, quindi un numero esponenziale. Ogni foglia corrisponde ad un assegnamento di tutte le variabili, mentre ogni nodo interno corrisponde ad un assegnamento parziale delle variabili. La radice corrisponde all'assegnamento di nessuna variabile, cioè alla formula iniziale. Per verificare tutti gli assegnamenti con una ricerca esaustiva dobbiamo visitare l'intero albero degli assegnamenti con un costo che è proporzionale al numero totale di nodi dell'albero.

Il backtracking ci aiuta a non esplorare tutto l'albero se non nei casi in cui è strettamente necessario. Infatti è possibile fermarsi prima (a parte quando troviamo una soluzione) quando ci accorgiamo che è inutile proseguire su un determinato cammino nell'albero in quanto sicuramente la formula non sarà soddisfatta dagli assegnamenti specificati in quel sottoalbero. Da qui il nome di "backtracking": torniamo indietro per provare altre strade. Vediamo come si applica tale tecnica all'esempio di cui prima.

Nel primo passo consideriamo la scelta per la variabile a. Questo ci porterà ad un nuovo problema, sottoproblema del precedente, nel quale abbiamo assegnato un valore ad a. Supponiamo di esplorare prima la scelta $a = 0$. La risultante formula sarà:

$$\begin{aligned} \phi(0, b, c, d) &= (\neg b + c + \neg d) \cdot (b) \cdot (\neg b) \cdot (1) \cdot (1) \\ &= (\neg b + c + \neg d) \cdot (b) \cdot (\neg b) \end{aligned}$$

Se, a partire da questo nuovo problema, consideriamo ora la scelta per la variabile b avremo un nuovo sottoproblema. Supponiamo di esplorare l'assegnamento $b = 0$,

avremmo che

$$\phi(0,0,c,d) = (c + \neg d) \cdot (0) \cdot (1) = 0$$

A questo punto è inutile procedere ad esplorare ulteriori assegnamenti a c e d: la formula rimarrà falsa. Conviene "tornare indietro" e provare altre scelte risparmiandoci così la visita di quel particolare sottoalbero.

La Figura 2.2 mostra il ragionamento fatto in precedenza. Il nodo radice corrisponde alla formula iniziale $\phi_1 = \phi$. La scelta $a = 0$ porta alla formula $\phi_2 = \phi(0, b, c, d)$, e l'ulteriore scelta $b = 0$ porta alla formula $\phi(0,0,c,d) = 0$. La figura mette in evidenza come le scelte $a = 0$ e $b = 0$ da sole determinano il valore di $\phi = 0$. La restante parte di quel sottoalbero può essere ignorata in quanto non conterrà nessun assegnamento che rende vera la formula.

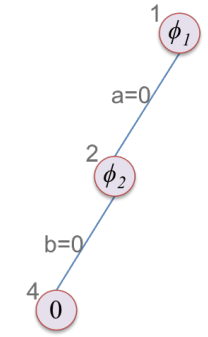

Figura 2.2: Valutazione parziale

Procedendo analogamente per le altre scelte, otteniamo l'albero mostrato nella Figura 2.3. In questo particolare esempio, abbiamo scoperto che la ϕ non è soddisfacibile senza esplorare tutto l'albero. Questo si traduce in un algoritmo più veloce rispetto alla ricerca esaustiva normale. Si ricordi però che in ogni caso la complessità rimane esponenziale.

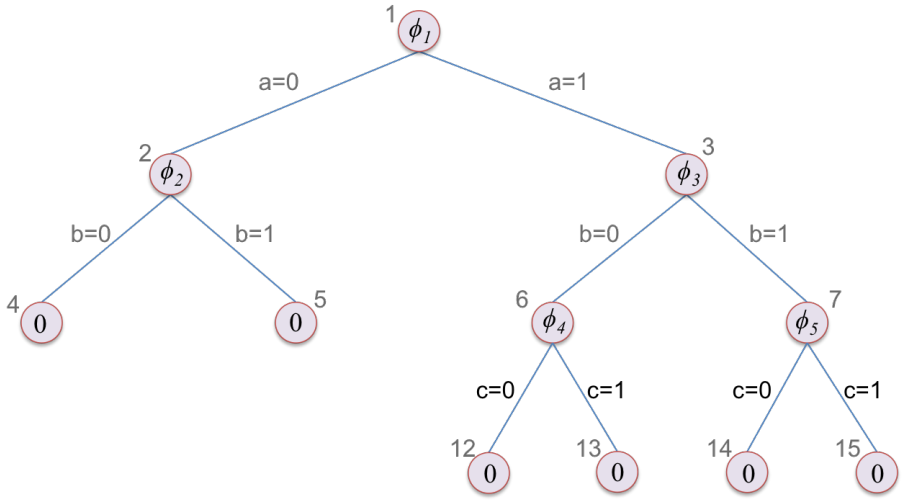

Figura 2.3: Albero di backtrack

2.4.2 Sudoku

Consideriamo ora un problema meno teorico e più conosciuto anche da chi non si occupa di informatica: il Sudoku. Partiamo da una versione più semplice, denominata LatinSquare. Nel problema LatinSquare abbiamo una scacchiera $n \times n$, in cui ogni cella può essere vuota oppure contenere un numero fra 1 e n. Il problema consiste nel riempire tutte le caselle inizialmente vuote con numeri compresi fra 1 e n in modo tale che in ogni riga e in ogni colonna non ci siano numeri ripetuti.

Osserviamo che se la scacchiera è completamente vuota è facile trovare una soluzione: nella prima riga possiamo inserire la sequenza $\langle 1, 2, \ldots, n-1, n \rangle$, e nelle successive

righe uno shift ciclico (nella riga i uno shift di $i-1$ posizioni, $i = 1, 2, \ldots, n$), come mostrato nella Figura 2.4 per il caso $n = 4$.

Se invece la scacchiera contiene già degli elementi potrebbe non essere possibile trovare un completamento che soddisfa il vincolo richiesto dal problema. In pratica la presenza di elementi iniziali riduce lo spazio delle potenziali soluzioni, ma lo può ridurre talmente tanto da renderlo vuoto. Un esempio è mostrato nella Figura 2.5: per non violare il vincolo del problema sulla seconda e terza riga, l'ultimo elemento deve necessariamente essere 1 e quindi ci sarebbe una violazione sulla quarta colonna. Per questa istanza non esiste un completamento.

Data una configurazione iniziale, in generale, per stabilire se la scacchiera può essere completata non sappiamo far meglio di una ricerca esaustiva; infatti il problema LatinSquare è NP-hard[1].

Il Sudoku è una versione più complessa di LatinSquare: la scacchiera del Sudoku contiene n^2 sottoscacchiere di dimensione $n \times n$, quindi è una scacchiera di $n^2 \times n^2$ caselle che devono essere riempite usando i numeri da 1 a n^2, in modo tale che in ogni riga, in ogni colonna ed in ogni sottoscacchiera $n \times n$, non ci siano mai numeri ripetuti. La Figura 2.6 mostra, per il caso $n = 3$, che è quello utilizzato nelle riviste di enigmistica, una scacchiera vuota, una configurazione legale ed una configurazione che vìola le 3 proprietà: nella riga, colonna e sottoschachiera evidenziate si ripete un 5.

1	2	3	4
4	1	2	3
3	4	1	2
2	3	4	1

Figura 2.4: Un esempio di LatinSquare per $n = 4$

1	2		4
4	3	2	
2	4	3	

Figura 2.5: Un esempio di LatinSquare per $n = 4$

[1] C. J. Colbourn. The complexity of completing partial latin square. *Discrete Applied Mathematics*, 8(1):25–30, 1984

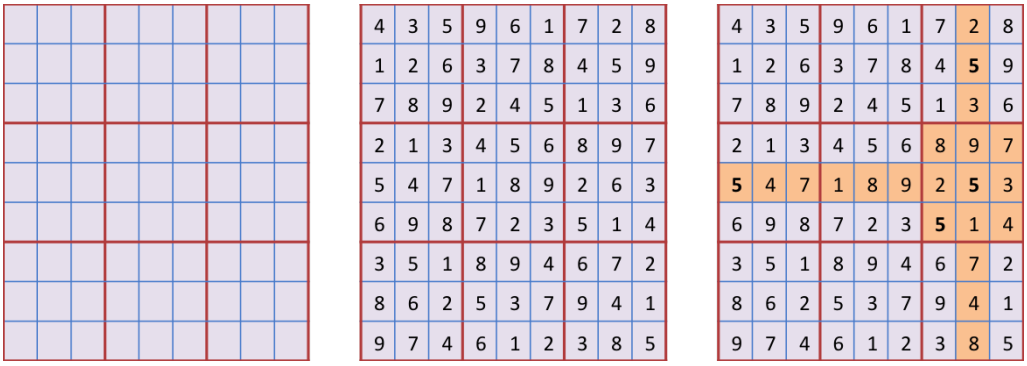

Figura 2.6: Scacchiere Sudoku per $n = 3$

Anche per il Sudoku è facile trovare una soluzione se la scacchiera è vuota; tuttavia la soluzione è leggermente meno immediata per via del vincolo sulle sottoscacchiere. Una soluzione, che chiameremo *soluzione base*, è la seguente: si riempie la prima sottoscacchiera con i numeri da 1 a n^2 procedendo prima per colonne e poi per righe; le altre sottoscacchiere verranno riempite con una permutazione fatta sia sulle righe che sulle colonne.

Più formalmente, chiameremo schacchiera "madre" la scacchiera del Sudoku. La scacchiera madre contiene n^2 sottoscacchiere che numeriamo partendo dall'alto verso il basso e da sinistra verso destra (quindi la sottoscacchiera 1 è quella in alto a sinistra, la sottoscacchiera n^2 quella in basso a destra. Nella prima "riga" di sottoscacchiere troviamo le sottoscacchiere numerate da 1 a n. Nella seconda quelle da $n+1$ a $2n$, e così via, fino all'ultima riga dove troviamo le sottoscacchiere dalla $(n-1)n+1$ a n^2.

Per riempire le n^2 sottoscacchiere procediamo in questo modo. Iniziamo riempiendo la prima colonna della scacchiera madre con i numeri da 1 a n. Quindi procediamo a riempire le colonne dalla 2 alla n della scacchiera madre con degli shift ciclici di n posizioni. La Figura 2.7 mostra questi due passi per il caso $n = 3$.

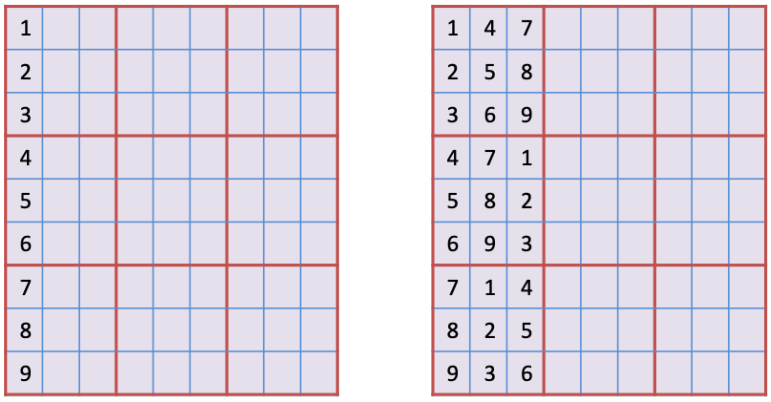

Figura 2.7: Prime n colonne della soluzione *base* del SUDOKU $n = 3$

Abbiamo così riempito la prima sottoscacchiera di ogni riga, cioè le sottoscacchiere numero $1, n+1, 2n+1, \ldots, (n-1n)n+1$. A questo punto procediamo a rimpiere le restanti sottoscacchiere operando un shift ciclico diagonale sulla scacchiera immediatamente alla sinistra. Lo shift ciclico diagonale si ottiene spostando l'elemento (i, j) nella posizione $(i+1, j+1)$ all'interno della sottoscacchiera. La Figura 2.8 mostra un esempio di shift ciclico diagonale.

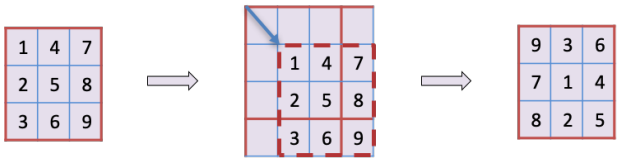

Figura 2.8: Shift ciclico diagonale.

Quindi, per ogni riga, la "prossima" scacchiera, numero i, viene riempita con lo shift ciclico diagonale della scacchiera precedente, numero $i-1$.

La Figura 2.9 mostra il risultato finale per il caso $n = 3$.

Come per il LATINSQUARE, anche per il SUDOKU, la presenza di numeri iniziali può rendere il problema non risolvibile, e per scoprirlo non sappiamo fare meglio di una ricerca esaustiva: non dovrebbe sorprendere il fatto che anche il SUDOKU è un problema $\mathcal{NP}$-hard.

Lemma 2.4.1 LATINSQUARE$\leq_P$SUDOKU.

DIMOSTRAZIONE. Consideriamo un'istanza del problema LATINSQUARE. Indichiamo con $c_1, \ldots, c_n$ le n colonne della scacchiera. Costruiamo la seguente istanza del SUDOKU per una scacchiera di dimensione $n^2 \times n^2$. Partiamo dalla soluzione base costruita come spiegato in precedenza da una scacchiera vuota. Individuiamo le colonne delle sottoscacchiere da 1 a n che contengono i numeri da 1 a n e sostituiamo con $c_1, \ldots, c_n$ tali colonne.

1	4	7	9	3	6	5	8	2
2	5	8	7	1	4	6	9	3
3	6	9	8	2	5	4	7	1
4	7	1	3	6	9	2	5	8
5	8	2	1	4	7	3	6	9
6	9	3	2	5	8	1	4	7
7	1	4	6	9	3	8	2	5
8	2	5	4	7	1	9	3	6
9	3	6	5	8	2	7	1	4

Figura 2.9: Permutazione colonne sottoscacchiere

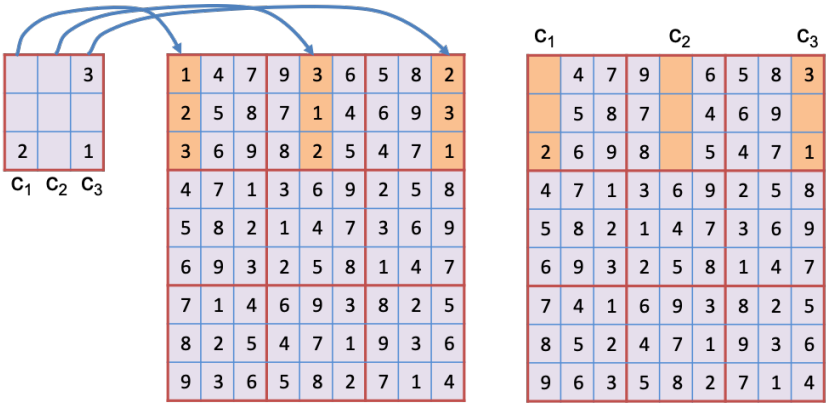

Figura 2.10: Istanza SUDOKU da istanza LATINSQUARE.

La Figura 2.10 mostra con un esempio per il caso $n = 3$ la costruzione dell'istanza del problema SUDOKU. A sinistra c'è un'istanza del problema LATINSQUARE, con le colonne c_1, c_2 e c_3. Nel centro c'è la soluzion base e sono state individuate le colonne con i numeri da 1 a n. A destra l'istanza del SUDOKU in cui le colonne c_1, c_2 e c_3 sono state inserite nelle prime 3 sottoscacchiere, nelle colonne individuate nel passo percedente.

La trasformazione descritta sopra può essere fatta in tempo polinomiale.

Per come è stata costruita l'istanza del SUDOKU si ha che nelle caselle vuote delle colonne c_i è possibile inserire solo i numeri da 1 a n. Pertanto non rimane che risolvere il problema del SUDOKU, chiamando l'oracolo. Le colonne c_i forniscono una soluzione anche all'istanza iniziale del problema LATINSQUARE: infatti la proprietà di non ripetizione sulle righe e sulle colonne della scachiera del SUDOKU, garantisce che non ci siano ripetizioni, sia sulle righe che sulle colonne, nei numeri contenuti nelle colonne c_i. Pertanto le colonne c_i della soluzione del SUDOKU sono una soluzione che completa l'istanza iniziale del LATINSQUARE. □

Il problema LATINSQUARE è $\mathcal{NP}$-hard. Quindi anche il problema del SUDOKU è $\mathcal{NP}$-hard.

Poichè non esiste una soluzione efficiente per il SUDOKU possiamo utilizzare la ricerca esaustiva. Specificheremo una potenziale soluzione con una sequenza σ di n^4 numeri:

$$\sigma = [\sigma_1, \sigma_2, \ldots, \sigma_{n^4}]$$

ordinati per righe (cioè i primi n^2 corrispondono alla prima riga della scacchiera, i successivi alla seconda e così via). Proseguiremo la spiegazione considerando il caso $n = 3$ per il quale σ ha 81 elementi. Per trovare una soluzione consideriamo tutte le possibili sequenze di 81 elementi, in cui ogni elemento è un numero fra 1 e 9. Una sequenza è una soluzione del SUDOKU, se soddisfa i vincoli del problema. Si noti che quando partiamo da una soluzione parziale, cioè quando alcuni numeri della sequenza sono già fissati, dovremo considerare meno possibilità.

Per verificare una sequenza dovremo controllare tutte le righe, tutte le colonne, e tutte le sottoscacchiere 3×3. Tuttavia quando troviamo una violazione di un vincolo in una parte della sottosequenza possiamo scartare in un solo colpo tutte le sequenze che hanno quella particolare sottosequenza. Ad esempio tutte le sequenze che inziano con $1, 2, 3, 4, 5, 6, 7, 8, 9, 3$ non sono una soluzione in quanto ci sono due 3 nella prima sottoscacchiera 3×3.

Anche per il SUDOKU, come fatto per SAT, possiamo usare un albero per rappresentare tutte le possibili sequenze. In questo caso l'albero è leggermente più complicato; in realtà è solo più grande, il che lo rende più complicato da disegnare. Infatti è un albero che ha 81 livelli, ognuno dei quali corrisponde ad una casella della scacchiera, ed ogni nodo dell'albero ha 9 figli, che corrispondono ai 9 possibili valori che possiamo inserire nelle caselle.

Ogni arco dell'albero rappresenta un assegnamento di uno specifico valore ad una specifica casella della scacchiera. Ogni nodo interno dell'albero corrisponde ad una sequenza parziale, data dagli assegnamenti dalla radice a quel nodo. Ogni foglia rappresenta una possibile sequenza. L'albero è enorme: ha 9^{81} foglie, e 9^{81} è un numero estremamente grande:

$$9^{81} = 196627050475552913618075908526912116283103450944214766927315415537966391196809.$$

Con un computer molto veloce, ad esempio capace di controllare $4 \cdot 10^9$ sequenze al secondo[2] avremmo bisogno di $9^{81}/(4 \cdot 10^9)$, cioè

$$4915676261888822840451897713172802907077586273605369173182885388491$$

secondi, il che significa più di 24277016742770 anni.

[2] Un computer con una CPU da 4GhZ può eseguire $4 \cdot 10^9$ istruzioni al secondo. Per controllare una sequenza non basta una singola istruzione.

Usando il backtracking possiamo esplorare lo spazio delle soluzioni più efficientemente.

2.5 *Esercizio di programmazione*

Scrivere un programma che utilizza il backtracking per risolvere il SUDOKU.

```
/* Il codice di un programma Java e' disponibile nel sito del corso */
```

2.6 *Branch and bound*

2.6.1 TSP

L'idea di base del backtracking può essere estesa a problemi di ottimizzazione. Il problema della soddisfacibilità di formule booleane è un problema decisionale, cioè un problema in cui si deve rispondere solo sì (se esiste una soluzione) oppure no (se non esiste una soluzione). Ogni soluzione è equivalente ad ogni altra soluzione e l'unica domanda che ci si pone è: esiste (almeno) una soluzione al problema? Nei problemi di ottimizzazione invece ogni soluzione ha un valore ed il problema è quello di trovare la soluzione ottima (un minimo o un massimo).

Come esempio considereremo il problema del commesso viaggiatore (Traveling Salesman Problem, TSP) già introdotto nel capitolo precedente: abbiamo un grafo $G = (V, E)$ con dei costi (distanze) associati ad ogni arco; ogni nodo rappresenta un luogo (città) ed ogni arco il costo per spostarsi da una città all'altra o la distanza fra le città; il commesso viaggiatore deve trovare un giro del grafo (cioè di tutte le città) in modo tale da visitare ogni città esattamente una volta e viaggiando/spendendo il meno possibile.

In questo problema l'albero delle decisioni rappresenta tutti i possibili "giri" del grafo. Poichè dobbiamo visitare ogni nodo esattamente una volta non ha importanza da quale nodo partiamo; detto in altre parole, possiamo scegliere un qualsiasi nodo come punto di partenza. L'albero delle decisioni rappresenta tutti i possibili cammini: ogni nodo dell'albero rappresenterà un cammino parziale dal nodo di partenza ad un altro nodo v ed i suoi figli saranno tutti i possibili nodi raggiungibili da v che non sono stati ancora visitati.

Consideriamo ad esempio il grafo mostrato nella Figura 2.11. Scegliendo di partire dal nodo a, le possibili scelte per il prossimo nodo sono b, c, d ed e. Da b possiamo raggiungere a, c, ed e, e così via. Scartando cammini che ci portano in nodi già visitati, possiamo costruire l'albero "dei percorsi" mostrato nella Figura 2.12.

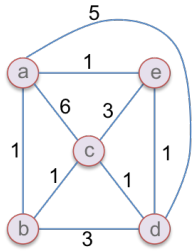

Figura 2.11: Grafo

Ovviamente tale albero ha un numero di nodi esponenziale in n, dove n è il numero di nodi nel grafo G. La Figura riporta solo una parte dell'albero evidenziando due possibili soluzioni (giri) una con costo 5 e l'atra con costo 12. I nodi contrassegnati con una X rossa sono percorsi che formano un ciclo e quindi non sono soluzioni del problema. Esplorare tutto l'albero costa tempo esponenziale in n.

Usando la stessa idea che ha portato un miglioramento nel caso del backtracking possiamo evitare di visitare parte di questo enorme albero se riusciamo in qualche modo a dire che in un determinato sottoalbero non c'è una soluzione ottima e quindi è inutile andare a cercarla lì. Per fare questo notiamo che stiamo cercando un valore minimo (il giro con distanza totale minima). Pertanto se riusciamo a dire che tutti i percorsi che possiamo individuare a partire da un determinato nodo dell'albero hanno un costo totale superiore al costo di un altro percorso già individuato possiamo concludere che in quel sottoalbero non ci sono soluzioni ottime e quindi possiamo evitare di visitarlo.

Un limite ovvio e semplice da ottenere è il costo del cammino parziale che corrisponde al nodo stesso. Un nodo interno dell'albero rappresenta un determinato percorso

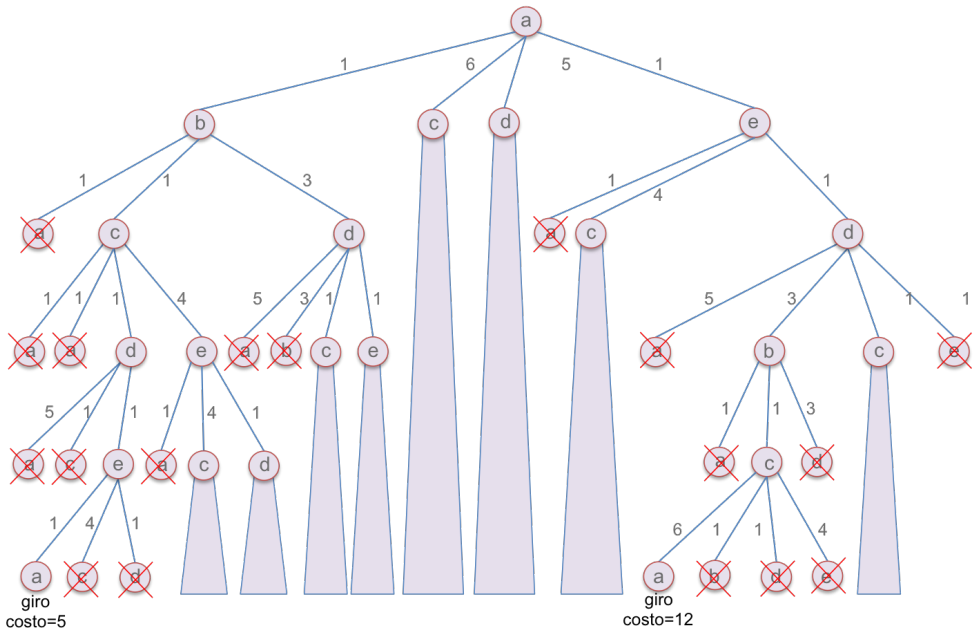

Figura 2.12: Albero dei percorsi

parziale α dal nodo di partenza, che nel nostro esempio è a, a un nodo $v = v(\alpha)$ che è il nodo al quale si arriva tramite il percorso parziale. Il costo del percorso α è chiaramente un limite inferiore per tutti i giri che hanno α come parte iniziale.

Inoltre possiamo ottenere dei limiti inferiori più alti osservando che il costo di un qualsiasi prosieguo β che permetta di completare α in un giro, dovrà essere la somma di almeno:

- il costo dell'arco con costo minimo da a ad un nodo non presente in α (perchè dobbiamo finire il giro in a, quindi dobbiamo prima o poi ritornare in tale nodo).
- costo dell'arco con costo minimo da v ad un nodo non presente in α (perchè qualunque percorso che estende α ha un arco dal nodo v ad un nodo $w \notin \alpha$).
- costo di un albero minimo ricoprente dei nodi non presenti in α (perchè dobbiamo visitare tutti i nodi ancora non visitati).

Una volta stabilito un limite inferiore al costo di un qualsiasi giro che è un completamento del percorso parziale α, se tale costo è più alto del costo di un giro già noto, possiamo evitare di visitare il sottoalbero radicato nel nodo $v(\alpha)$.

Riprendendo l'esempio precedente e sfruttando la tecnica di branch and bound, riusciamo a risolvere il problema visitando una parte molto più piccola dell'albero, come mostrato nella Figura 2.13. In questo esempio è stato usato il costo del cammino parziale già costruito, come limite inferiore al costo totale dei giri che si possono ottenere a partire dal cammino parziale. Dopo aver trovato il primo giri di costo 5, che in questo esempio è il costo ottimale, potremo evitare di esplorare i sottoalberi di quei nodi in cui il limite inferiore è maggiore o uguale a 5. La figura mette in evidenza il fatto che usando la tecnica di branch and bound eviteremo di esplorare 7 sottoalberi, di cui 2 che

partono dal livello 1 dell'albero, 1 che parte da livello 2 ed altri 4 che partono dal livello 3.

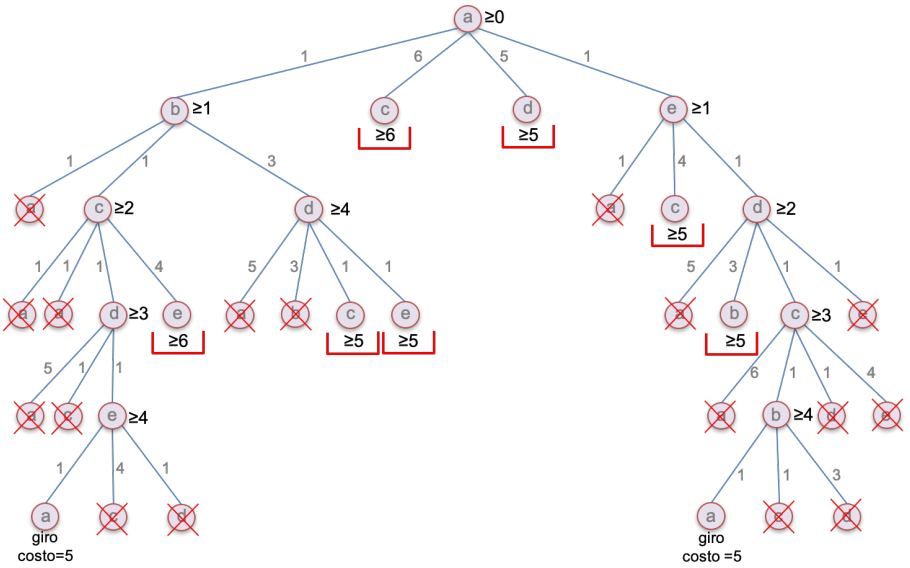

Figura 2.13: Albero dei percorsi branch-and-bound

2.7 *Scelta dell'albero di backtracking*

La tecnica di backtracking, come abbiamo visto, si basa sulla costruzione di un albero che rappresenta tutte le possibili soluzioni. La costruzione di tale albero può essere fatta in vari modi ed è importante scegliere quello più utile per poter poi implementare il backtrack stesso. Consideriamo ad esempio il problema VertexCover. Ricordiamo il problema: dato un grafo $G = (V, E)$ e un intero k, vogliamo sapere se esiste un insieme di nodi $S \subseteq V$ di al massimo k nodi tale che per ogni arco $(u, v) \in E$, risulta che almeno un nodo fra u e v appartiene a S. Un insieme S con tale caratteristica è un *ricoprimento (degli archi) di G*, cioè un *vertex cover*. Dunque nel problema VertexCover le possibili soluzioni sono tutti i possibili sottoinsiemi di V con al più k nodi. Per semplicità consideriamo una variante del problema in cui ci chiediamo se esistono ricoprimenti di *esattamente k* nodi. Questo per far sì che solo le foglie dell'albero rappresentino soluzioni e non pure i nodi interni; ma non è importante per quello di cui stiamo discutendo. Come possiamo costruire un albero che rappresenta tali sottoinsiemi?

Consideriamo questa idea: la radice dell'albero rappresenta l'insieme vuoto. Ogni livello dell'albero rappresenta l'inserimento di un nuovo nodo nell'insieme S. L'albero ha k livelli. La radice ha quindi esattamente n figli in quanto nell'insieme vuoto possiamo inserire un qualsiasi nodo. I nodi del primo livello invece avranno esattamente $n - 1$ figli in quanto essi rappresentano un insieme di un nodo e quindi quel nodo non può essere inserito di nuovo. Analogamente i nodi del secondo livello avranno $n - 2$, ed in generale i nodi del livello i avranno $n - i$ figli. Le foglie di questo albero rappresentano tutti i possibili sottoinsiemi di k nodi.

Tuttavia, questa scelta non ci aiuta per il backtrack. Infatti per mettere in pratica la strategia di backtrack, quando ci troviamo in un nodo a dell'albero di backtracking dobbiamo riuscire a dire se nel sottoalbero radicato in a ci sono soluzioni al problema. Non è semplice farlo: di fatto significa risolvere un sottoproblema che consiste nello stabilire se con i nodi rimanenti è possibile ricoprire il sottografo formato dagli archi non ancora ricoperti.

Consideriamo ora quest'altra idea: la radice dell'albero rappresenta l'insieme V mentre ogni livello rappresenta la cancellazione di un nodo dall'insieme precedente. Analogamente al caso precedente la radice avrà esattamente n figli, in quanto possiamo eliminare uno qualsiasi dei nodi di V, mentre i nodi dei livelli successivi avranno meno figli, $n - i$ al livello i, in quanto ci sono meno nodi da poter eliminare. L'albero in questo caso avrà $n - k$ livelli perchè da un insieme di n nodi vogliamo arrivare ad un insieme di k nodi. Come nel caso precedente le foglie rappresentano tutti i sottoinsiemi di taglia k.

Consideriamo ad esempio il grafo nella Figura 2.14. La Figura 2.15 mostra l'albero.

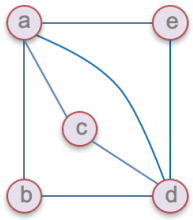

Figura 2.14

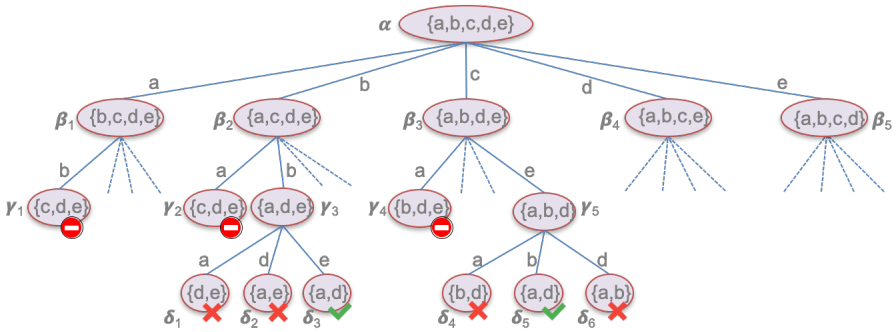

Figura 2.15: Albero di backtrack

La radice α corrisponde all'insieme di tutti i nodi che è banalmente un insieme ricoprente. Da questo insieme possiamo provare a togliere un nodo alla volta. Il primo livello corrisponderà a sottoinsiemi di 4 vertici del grafo. Ad esempio il nodo β_1 corrsiponde al sottoinsieme $\{b, c, d, e\}$ (abbiamo tolto il nodo a), il nodo β_2 corrsiponde al sottoinsieme $\{a, c, d, e\}$ (abbiamo tolto il nodo b), e così via.

Ora però se consideriamo un nodo x interno all'albero abbiamo un modo semplice per dire che nel sottoalbero radicato in x non ci sono ricoprimenti: se esiste un arco $e = (u, v)$ per il quale né u né v sono nell'insieme rappresentato da x, allora non è possibile ricoprire e. Quindi possiamo fermare al nodo a l'esplorazione dell'albero senza scendere nel sottoalbero in esso radicato. Ad esempio il nodo γ_1 rappresenta l'insieme $\{c, d, e\}$. Questo insieme non è ricoprente in quanto l'arco (a, b) non è ricoperto; pertanto nessun sottoinsieme di $\{c, d, e\}$ può ricoprire il grafo. Dunque l'esplorazione del sottoalbero radicato in γ_1 è inutile.

Notiamo però che in questo albero ci sono sottoinsiemi che sono rappresentati da più nodi. Ad esempio i nodi γ_1 e γ_2 rappresentano entrambi il sottoinsieme $\{c, d, e\}$. Questo significa che nell'albero ci sono dei percorsi ridondanti che ovviamente sono un problema: la visita dell'albero richiede più tempo del necessario.

Una rappresentazione che evita questo problema è la seguente: ogni livello dell'albero

rappresenta un vertice del grafo e ogni nodo dell'albero ha due figli che rappresentano, rispettivamente, la non appartenenza e l'appartenza all'insieme del vertice. In pratica è la stessa che abbiamo usato per il problema SAT, con i vertici al posto delle variabili ed i valori booleani che indicano l'appartenza all'insieme. La Figura 2.16 mostra l'albero.

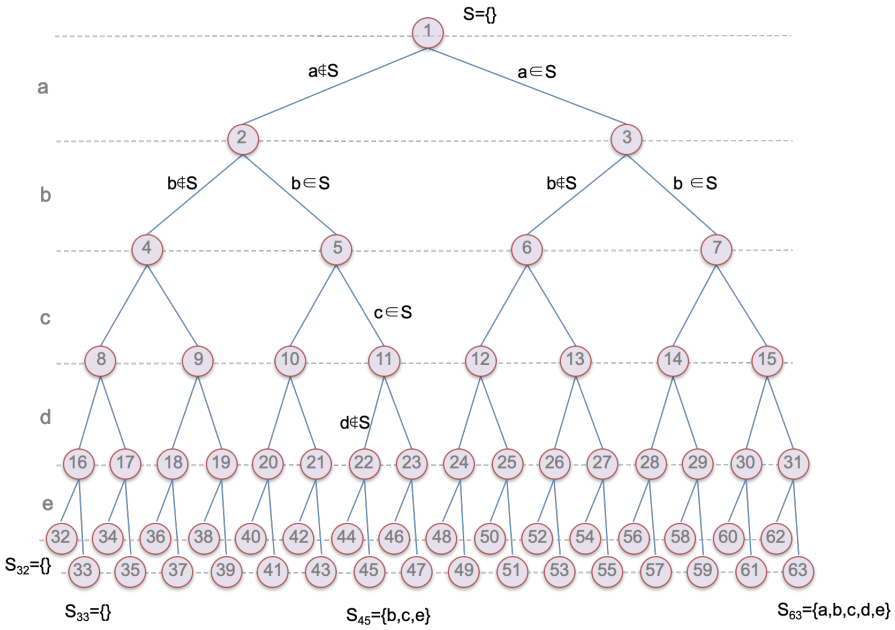

Figura 2.16: Albero completo

Le 32 foglie di questo albero corrispondono ai $2^5 = 32$ possibili sottoinsiemi che si possono avere con i 5 vertici del grafo. Ogni nodo interno rappresenta un insieme di sottoinsiemi con caratteristiche in comune. Ad esempio il nodo 4 rappresenta tutti i sottoinsiemi che non contengono né il vertice *a* né il vertice *b*. Quindi, come osservato in precedenza, nessuno dei sottoinsiemi rappresentati dal nodo 4 puòessere un insieme ricoprente. Pertanto è inutile visitare il sottoalbero radicato in 4.

Inoltre possiamo osservare che poichè l'insieme ricoprente può contenere al massimo k nodi non possiamo usare più di k volte l'arco che collega un padre al suo figlio destro. Duqnue ogni volta che raggiungiamo un nodo usando k archi "destri" ci fermiamo. Ad esempio per $k = 2$ avremmo l'albero di backtrack mostrato nella Figura 2.17.:

Dunque il modo in cui si costruisce l'albero di backtracking è importante in quanto determina l'efficacia della strategia. Ovviamente la cosa dipende dal problema che si considera e quindi le specifiche scelte vanno fatte caso per caso.

2.8 *Sfruttare le caratteristiche del problema*

In alcuni casi è possibile sfruttare le specifiche caratteristiche del problema o dell'input al problema per ottenere dei miglioramenti. Nel seguito vedremo alcuni esempi.

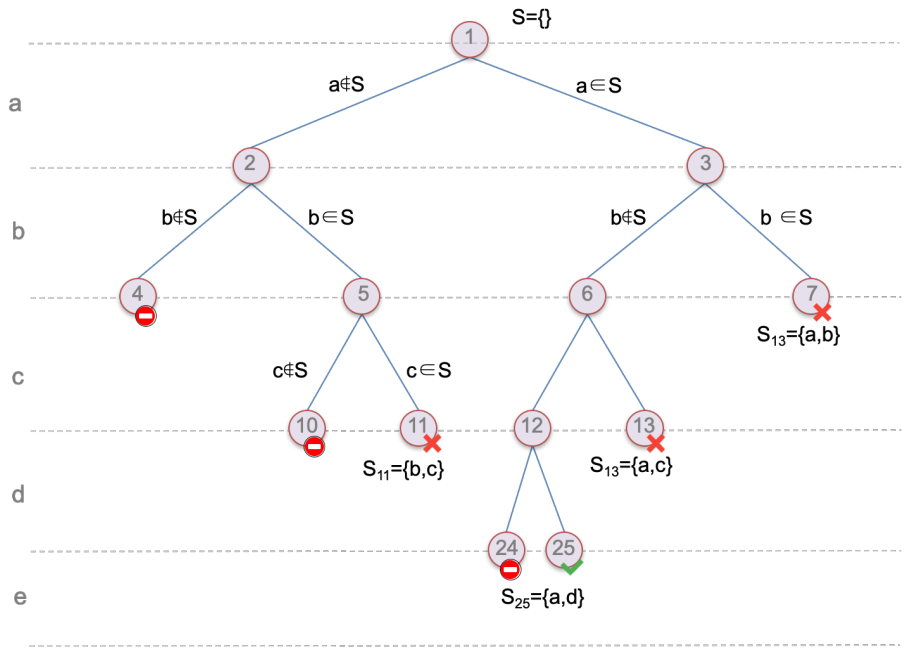

Figura 2.17: Albero di backtrack per $k = 2$

2.8.1 *Vertex Cover*

Riprendiamo nuovamente il problema VertexCover. Poichè il problema è difficile, la complessità della soluzione è una funzione esponenziale della taglia dell'input. Tuttavia in tale funzione gioca un ruolo fondamentale anche il parametro k. Usando un approccio a forza bruta, possiamo risolvere il problema semplicemente considerando tutti i possibili sottoinsiemi di V di taglia k. I sottoinsiemi di taglia k sono esattamente $\binom{n}{k}$ e controllare se un sottoinsieme è un ricoprimento degli archi richiede tempo $O(kn)$. Pertanto l'algoritmo a forza bruta richiede tempo $O(kn\binom{n}{k}) = O(kn^{k+1})$. Pertanto se k è fissato la complessità della soluzione a forza bruta è polinomiale! In altre parole l'intrattabilità del problema si manifesta solo quando k cresce con n. Quindi se dobbiamo risolvere il problema per $k = 2$, anche l'algoritmo a forza bruta è "efficiente": la sua complessità è $O(n^3)$. Discorso analogo per altri valori di k; ovviamente più k diventa grande, più l'algoritmo diventa inefficiente.

Infatti, anche per valori relativamente piccoli di k un tempo di esecuzione $O(kn^{k+1})$ può essere molto grande. Ad esempio per $n = 1000$ e $k = 10$, ignorando le costanti nascoste nella notazione asintotica e assumendo che il computer che esegue il programma esegua un milione di istruzioni di alto livello per ogni secondo, occorrerebbero 10^{28} secondi[3]. E questo nonostante il valore di k sia relativamente piccolo, caso per il quale avevamo detto che il problema poteva essere trattabile. Tuttavia se al posto della forza bruta usiamo un approccio più scaltro, possiamo ottenere un algoritmo che, sebbene, ovviamente, esponenziale come quello a forza bruta, risulta più efficiente per valori piccoli di k. Vedremo un algoritmo il cui tempo di esecuzione è $O(2^k kn)$. Si noti come in questo caso l'esponenzialità del tempo di esecuzione sia dovuta a k che compare come esponente di una costante; nell'approccio a forza bruta k compare

[3] 1 secolo dura meno di 10^{10} secondi; 10^{28} è un tempo spaventosamente grande, ordini di magnitudine più grande dell'età dell'universo

come esponente di n. Questa differenza è sostanziale. Sebbene per k grandi il tempo di esecuzione può comunque essere incredibilmente grande, per k piccoli, anche al crescere di n il tempo di esecuzione può essere ragionevole. Ad esempio, per il caso considerato in precedenza, cioè $n = 1000$ e $k = 10$, usando lo stesso computer, il tempo di esecuzione diventa di circa 10 secondi. Al crescere di k ovviamente il tempo diventa comunque molto grande; tuttavia l'aver spostato la dipendenza esponenziale da k fuori dall'esponente di n ha creato una situazione nettamente migliore dal punto di vista pratico.

Vediamo l'algoritmo. Iniziamo con l'osservare che se un grafo ha un insieme ricoprente piccolo non può avere molti archi. Più formalmente abbiamo il seguente lemma.

Lemma 2.8.1 *Se $G = (V, E)$ ha un insieme ricoprente di taglia k allora $|E| \leq k(n-1)$, dove $n = |G|$ è il numero di nodi del grafo.*

DIMOSTRAZIONE. Sia S un insieme ricoprente di taglia k. Ogni nodo di S può "ricoprire" al massimo $n-1$ archi (ogni nodo può essere collegato al massimo a tutti gli altri $n-1$ nodi). Quindi S può ricoprire al massimo $k(n-1)$ archi. D'altra parte per definizione di insieme ricoprente si ha che tutti gli archi sono ricoperti da S, quindi il numero totale di archi non può essere superiore a $k(n-1)$. □

Il Lemma 2.8.1 permette di assumere che il grafo da esaminare ha al massimo $k(n-1)$ archi. Infatti possiamo controllare la cardinalità di E e se è maggiore di $k(n-1)$ risolviamo il problema immediatamente rispondendo che G non ha un insieme ricoprente di taglia k.

L'idea alla base dell'algoritmo che descriveremo fra poco è molto semplice ed elegante. Consideriamo un arco $e = (u, v)$ di G. Sia S un qualsiasi insieme ricoprente S di taglia k; almeno uno fra u e v deve appartenere a S. Supponiamo che $u \in S$. Allora, se cancelliamo u e tutti gli archi incidenti su u, deve essere possibile ricoprire gli archi rimanenti con un insieme di al massimo $k-1$ nodi. Cioè, se consideriamo il grafo $G - \{u\}$, ottenuto da G cancellando u e tutti gli archi ad esso incidenti, si ha che deve esistere un insieme ricoprente di taglia al massimo $k-1$. Stesso ragionamento può essere fatto nel caso in cui sia v ad appartenere a S. Possiamo formalizzare quanto appena detto con il seguente lemma.

Lemma 2.8.2 *Sia $e = (u, v)$ un arco di G. Il grafo G ha un insieme ricoprente di taglia k se e solo se almeno uno dei grafi $G \setminus \{u\}$ e $G \setminus \{v\}$ ha un insieme ricoprente di taglia $k-1$.*

DIMOSTRAZIONE. Assumiamo che G abbia un insieme ricoprente S di taglia k. Allora S deve contenere almeno uno fra u e v; supponiamo che sia u ad appartenere a S. Si consideri l'insieme $S' = S - \{u\}$; tale insieme è un insieme ricoprente per $G - \{u\}$; infatti tutti gli archi non incidenti ad u sono ricoperti da nodi di S'. Inoltre $|S'| = k-1$. Se u non appartiene a S allora v appartiene ad S; in tal caso $S'' = S - \{v\}$ è un insieme ricoprente per $G - \{v\}$.

Viceversa, supponiamo che almeno uno dei grafi $G - \{u\}$ e $G - \{v\}$ abbia un insieme ricoprente S di taglia $k-1$. Supponiamo sia $G - \{u\}$ (se fosse $G - \{v\}$ basterà sostituire u

con v). Per definizione, S ricopre tutti gli archi di $G - \{u\}$; se consideriamo $S' = S \cup \{u\}$ si ha che S' ricopre G in quanto gli archi che G ha in più rispetto a $G - \{u\}$ sono quelli incidenti su u. □

Il Lemma 2.8.2 suggerisce in modo naturale il seguente algoritmo ricorsivo.

Algorithm 3: RecursiveVC(G, k)

```
if G = (V, E) non ha archi (caso base) then
    restitusci l'insieme vuoto S = {}
if |E| > k(n − 1) then
    G non ha un insieme ricoprente di taglia ≤ k
else
    Sia e = (u, v) un arco di G.
    Risolvi RecursiveVC(G − {u}, k − 1) e RecursiveVC(G − {v}, k − 1)
    if nessuno dei due sottoproblemi ha soluzione then
        G non ha un insieme ricoprente di taglia ≤ k
    else
        Uno dei due, diciamo G − {u}, ha un ricoprimento T
        T ∪ {u} è un ricoprimento di taglia ≤ k di G
        (Se fosse stato G − {v}, il ricoprimento sarebbe stato T ∪ {v})
```

Rimane da vedere la complessità dell'algoritmo. Ogni chiamata dell'algoritmo con parametro k produce due chiamate ricorsive con parametro $k-1$. Pertanto ci saranno in totale 2^{k+1} chiamate ricorsive e in ogni chiamata ricorsiva il tempo speso è di $O(kn)$. Possiamo analizzare formalmente il tempo necessario usando una relazione di ricorrenza. Se denotiamo con $T(n,k)$ il tempo necessario a risolvere il problema su un grafo di n nodi per una taglia dell'insieme ricoprente di k, avremo che esiste una costante c per la quale

$$\begin{aligned} T(n,k) &\leq 2T(n-1,k-1) + ckn, \\ T(n,1) &\leq cn. \end{aligned}$$

Possiamo procedere per induzione su k e provare che $T(n,k) \leq c \cdot 2^k kn$. Per $k = 1$ il limite è rispettato; assumiamo che sia vero per $k-1$ e proviamo che è vero per k:

$$\begin{aligned} T(n,k) &\leq 2T(n-1,k-1) + ckn, \\ &\leq 2c \cdot 2^{k-1}(k-1)n + ckn, \\ &= c \cdot 2^k kn - c \cdot 2^k n + ckn, \\ &\leq c \cdot 2^k kn. \end{aligned}$$

Riassumendo, l'algoritmo RecursiveVC fornisce un buon miglioramento dal punto di vista pratico. Tuttavia rimane esponenziale e nessun algoritmo esponenziale può rimanere efficiente al crescere dei parametri. Ad esempio per $k = 40$ anche l'algoritmo RecursiveVC, fatto girare sullo stesso computer usato in precedenza, richiederà un cospicuo numero di anni per terminare.

2.8.2 *Independent Set*

Consideriamo adesso un altro problema per illustrare un caso in cui si riesce a sfruttare un vincolo sulla struttura dell'input per "ammorbidire" l'intrattabilità del problema. Il problema che consideriamo è IndependentSet che, come abbiamo visto nel capitolo precedente, è un problema $\mathcal{NP}$-completo. Ricordiamo il problema: dato un grafo $G = (V, E)$ e un intero k, vogliamo sapere se esiste un insieme di nodi $S \subseteq V$ di almeno k nodi tale che non ci sia nessun arco fra i nodi di S.

In questo caso vogliamo sfruttare una restrizione sull'input. Al posto di considerare il problema su grafi qualsiasi restringiamo l'attenzione a grafi che sono alberi. La semplicità della struttura di un albero rispetto al caso generale di un grafo qualsiasi permette di fornire un algoritmo efficiente. Sebbene utilizzeremo il problema IndependentSet come esempio, l'approccio può essere sfruttato in generale; ovviamente non è detto che funzioni per qualsiasi problema.

Vediamo adesso l'algoritmo. Iniziamo con la seguente semplice osservazione: ogni arco $e = (u, v)$ del grafo G si ha che un qualsiasi insieme indipendente S di G può contenere al massimo uno fra u e v. Vogliamo sfruttare questa osservazione per progettare un algoritmo greedy che possa scegliere, dato un arco e, di inserire nell'insieme indipendente uno dei due nodi su cui e incide.

Per prendere la decisione greedy sfrutteremo il fatto che il grafo sul quale operiamo è un albero. Poichè il grafo è un albero esiste almeno una foglia. Sia $e = (u, v)$ un arco tale che v è una foglia. Se includiamo v nell'insieme indipendente non potremo inserire u; ma u è l'unico altro nodo che non potremo inserire nell'insieme in quanto v è collegato solo ad u. Analogamente, se inseriamo u nell'insieme indipendente non potremo inserire v; in questo caso però potrebbero esserci altri nodi (l'eventuale padre di u e eventuali altri figli di u) che non potranno essere inseriti nell'insieme indipendente. Quindi inserire v nell'insieme indipendente non previene la possibilità di costruire un insieme indipendente massimale. Questo è l'ingrediente chiave per avere un algoritmo greedy che trova l'ottimo. Più formalmente abbiamo il seguente lemma.

Lemma 2.8.3 *Sia $T = (V, E)$ un albero e v una foglia di T. Allora esiste un insieme indipendente massimale che contiene v.*

Dimostrazione. Sia S un insieme indipendente massimale e sia $e = (u, v)$ l'unico arco incidente su v. Poichè S è massimale almeno uno fra u e v deve essere contenuto in S; se così non fosse potremmo costruire l'insieme indipendente $S \cup \{v\}$ di cardinalità $|S| + 1$ contraddicendo il fatto che S è massimale. Se è proprio v ad appartenere ad S il lemma è soddisfatto. Se invece è u ad appartenere ad S possiamo costruire l'insieme indipendente $S' = S - \{u\} \cup \{v\}$, cioè togliere u ed inserire v, ottenendo un nuovo insieme indipendente che è di taglia massimale, in quanto $|S'| = |S|$, e contiene v. □

Il lemma precedente suggerisce un semplice algoritmo greedy, che individua le foglie dell'albero e le inserisce nell'insieme indipendente e ripete l'operazione dopo aver cancellato i nodi (e gli archi ad essi incidenti) collegati al nodo inserito. Si noti che poichè la cancellazione di archi può disconettere l'albero, durante l'esecuzione dell'algorimo potremmo ritrovarci con una foresta. Quanto detto riguardo un nodo

foglia continua ovviamente a valere; semplicemente avremo vari alberi che potremo trattare singolarmente. Anzi avremo più foglie da scegliere, almeno una per ogni albero. Pertanto l'algoritmo opererà su una foresta, ma non è necessario nessun accorgimento particolare.

Algorithm 4: GREEDYIS(F)

S={} è l'insieme indipendente inzialmente vuoto
while *F ha almeno un arco* **do**
  Sia $e = (u, v)$ un arco di F tale che v è una foglia.
  $S = S \cup \{v\}$
  Cancella da F i nodi u, v e tutti gli archi ad essi incidenti
return S

La correttezza dell'algoritmo deriva direttamente dal Lemma 2.8.3. Il lemma garantisce che per ogni foglia v esiste un insieme indipendente S massimale che la contiene. Chiaramente S non potrà contenere u, per cui il fatto che l'algoritmo lo cancelli non è un problema. Inoltre non è un problema nemmeno cancellare gli archi incidenti su u: infatti se da un lato la cancellazione di archi può solo portare ad insiemi più grandi, dall'altro, non c'è il rischio che questi archi cancellati possano far sì che in S vengano inseriti nodi fra i quali esiste un arco, in quanto u stesso non è in S.

Concludiamo con l'osservare che la scelta greedy permessa dalla struttura ad albero in realtà può funzionare anche in un grafo generico a patto che venga soddisfatta la proprietà cruciale garantita da una foglia: se v è una foglia collegata al padre da $e = (u, v)$ allora u è l'unico "vicino" di v. Quindi anche se il grafo non è un albero ma esiste un nodo v che ha un unico vicino u, allora possiamo inserire v nell'insieme indipendente che stiamo costruendo in quanto siamo sicuri che tale scelta greedy non preclude la possibilità di costruire un insieme massimale, cioè esiste un insieme massimale che include v. Quindi l'approccio greedy descritto per gli alberi può essere usato parzialmente anche su grafi generici quando si crea la situazione di un nodo "foglia".

Infine osserviamo che affinchè il tutto funzioni è necessario che l'implementazione dell'algoritmo permetta di trovare in modo efficiente ad ogni iterazione una foglia; ciò non è difficile da fare ma è chiaramente fondamentale.

2.8.3 *Insiemi indipendenti pesati*

Consideriamo adesso una generalizzazione del problema INDEPENDENTSET: WEIGHTEDIS, nel quale abbiamo dei pesi per ogni nodo e l'insieme indipendente non deve più massimizzare il numero di nodi ma la somma dei pesi dei nodi. Più formalmente, abbiamo un albero $T = (V, E)$ ed anche una funzione che associa ad ogni nodo $v \in V$ il peso w_v. Vogliamo trovare un insieme indipendente S nel grafo T, tale che il peso totale di S, $w(S) = \sum_{v \in V} w_v$, sia quanto più grande possibile.

Osserviamo innanzitutto che la presenza dei pesi rende la decisione sull'inserimento dei nodi più difficile. Infatti se riconsideriamo l'approccio usato per l'algoritmo greedy ci accorgiamo immediatamente che non funziona più: mentre con l'assenza di pesi

inserire il nodo foglia non precludeva la possibilità di trovare comunque un insieme massimale, con i pesi è facile verificare che ciò non è più vero. Consideriamo un arco $e = (u, v)$ in cui v è una foglia. Se includiamo v non possiamo includere u. Questo non è un problema se $w_u \leq w_v$. Se invece $w_u > w_v$ escludendo v ci potremmo ritrovare con un insieme indipendente con peso non massimale. La presenza dei pesi rende la decisione sul se includere o meno un nodo, più difficile da prendere usando solo informazioni locali senza guardare al resto del grafo.

Tuttavia anche con la presenza di pesi possiamo comunque dire qualcosa se sono coinvolte delle foglie: se un nodo u ha molti figli $v_1, v_2, \ldots$ che sono foglie, allora dovremo prendere la stessa decisione per tutte le foglie. Cioè o le includiamo tutte o le escludiamo tutte. In altre parole per il sottoalbero radicato in u ci sono solo due possibili scelte: o includiamo u, e quindi escludiamo tutti i suoi figli, oppure escludiamo u, ed in questo caso dobbiamo includere tutti i sui figli.

Questa osservazione permette di sviluppare un algoritmo di programmazione dinamica. Ricordiamo che la programmazione dinamica è utile quando il problema può essere scomposto in sottoproblemi e le soluzioni dei sottoproblemi permettono di costruire la soluzione al problema originale[4]. Per il prolema WeightedIS, i sottoproblemi possono essere definiti nel seguente modo. Possiamo fissare arbitrariamente una radice dell'albero, diciamo il nodo r. Fissare la radice significa che tutti gli archi saranno orientati per "allontanarsi" dalla radice, quindi per ogni nodo u, il padre di u, $p(u)$, è il nodo che precede u nel cammino dalla radice a u. Gli altri nodi collegati da un arco ad a u saranno figli di u e li denoteremo nel loro insieme con $children(u)$. Il nodo u e tutti i suoi discendenti formano un sottoalbero T_u radicato nel nodo u.

[4] Inoltre, l'efficienza dell'algoritmo di programmazione è migliore quando molti dei sottoproblemi si ripresentano spesso per cui è sufficiente risolverli solo una volta risparmiando così tempo quando si ripresentano; tutavia questo apsetto non è fondamentale nella discussione che stiamo facendo.

I sottoalberi rappresentano i sottoproblemi. L'albero T_r è il problema originale. Se $u \neq r$ è una foglia, allora T_u ha un solo nodo (il nodo u). Se invece u ha figli che sono foglie allora il sottoalbero T_u ha la forma che ci permette di applicare l'osservazione fatta in precedenza.

Quindi, per risolvere il problema usando la programmazione dinamica, operiamo partendo dalle foglie e risalendo nell'albero. Per ogni nodo u, abbiamo bisogno di risolvere il sottoproblema rappresentato dall'albero T_u dopo aver risolto i sottoproblemi radicati nei figli di u. Per costruire un insieme indipendente S di peso massimo per T_u, dobbiamo cosiderare i due possibili casi: includiamo u in S oppure non lo includiamo. Nel caso in cui non includiamo u, per i figli di u abbiamo la possibilità di includerli o meno, in funzione di se conviene oppure no. Questa osservazioni suggeriscono la definizione di due sottoproblemi per ogni sottoalbero T_u: il sottoproblema $\text{OPT}_{inc}(u)$, che denota il peso massimo di un insieme indipendente per T_u che *include* u e il sottoproblema $\text{OPT}_{esc}(u)$, che denota il peso massimo di un insieme indipendente per T_u che *esclude* u.

Vediamo ora come calcolare OPT_{inc} e OPT_{esc}. Per una foglia u si ha che $\text{OPT}_{inc}(u) = w_u$ e $\text{OPT}_{esc}(u) = 0$. Per tutti gli altri nodi si ha la seguente relazione di ricorrenza

$$\begin{aligned} \text{OPT}_{inc}(u) &= w_u + \sum_{v \in children(u)} \text{OPT}_{esc}(v) \\ \text{OPT}_{esc}(u) &= \sum_{v \in children(u)} \max\{\text{OPT}_{inc}(v), \text{OPT}_{esc}(v)\}. \end{aligned}$$

La relazione di ricorrenza definita poc'anzi si traduce immediatamente in un algoritmo di programmazione dinamica. L'unica accortenza che dobbiamo avere è quella di visitare l'albero in modo tale che quando calcoliamo OPT_{inc} e OPT_{esc} per un nodo u dobbiamo aver già calcolato gli stessi per i figli di u. Per garantire questo fatto è sufficiente visitare l'albero con una visita post-order.

Algorithm 5: DynProWIS(T)

Scegli un nodo $r \in T$ come radice.
for *tutti i nodi u di* T_r *in post-order* **do**
  if *u è una foglia* **then**
    $\text{OPT}_{inc}(u) = w_u$
    $\text{OPT}_{esc}(u) = 0$
  else
    $\text{OPT}_{inc}(u) = w_u + \sum_{v \in children(u)} \text{OPT}_{esc}(v)$
    $\text{OPT}_{esc}(u) = \sum_{v \in children(u)} \max\{\text{OPT}_{inc}(v), \text{OPT}_{esc}(v)\}$
return $\max\{\text{OPT}_{inc}(r), \text{OPT}_{esc}(r)\}$

Come per tutti gli algoritmi di programmazione dinamica il mero calcolo del valore della soluzione ottima restituito da DynProWIS(T) non fornisce anche una soluzione che ha quel valore. Ma è facile modificare l'algoritmo memorizzando ad ogni passo l'informazione sul se il nodo u è stato inserito o meno nell'insieme e quindi, una volta ottenuto il valore finale, riscostruire una soluzione con quel valore.

2.9 *Algoritmi di ricerca locale*

Un'altra tecnica di carattere generale che può essere utilizzata per affrontare problemi di ottimizzazione difficili caratterizzati da uno spazio delle possibili soluzioni enorme, sono gli algoritmi di ricerca (dell'ottimo) locale. L'idea alla base di questa tecnica è la seguente: partendo da una soluzione qualsiasi si cerca di migliorarla facendo pochi cambiamenti alla soluzione stessa. A tal fine si definiscono delle regole in base alle quali due soluzioni sono considerate vicine (cioè passare dall'una all'altra richiede pochi cambiamenti). In base a tali regoli, per ogni soluzione s si può definire il *vicinato* di s, come tutte quelle soluzioni s' che sono vicine ad s. Un generico algoritmo di ricerca locale può essere descritto tramite il seguente pseudocodice:

Algorithm 6: GenericLS

Sia s una soluzione qualsiasi
while *Esiste* s' *vicino di* s *con costo*(s') *migliore di costo*(s) **do**
  $s \leftarrow s'$.
return s

Come si può intuire dallo pseudocodice, un algoritmo di ricerca locale è estremamente semplice: basta definire una regola di vicinanza fra le soluzioni (che ovviamente dipende strettamente dal problema) ed implementare un ciclo in cui ad ogni iterazione si cerca di migliorare la soluzione attuale cercandone una migliore fra quelle nel vicinato. Dunque un grosso vantaggio è la semplicità di implementazione dell'algoritmo. Il rovescio della

medaglia è che per questo tipo di approccio è spesso difficile fornire delle garanzie sulla soluzione che l'algoritmo produrrà. Infatti l'approccio viene detto di "ricerca locale" in quanto fornisce una soluzione che è ottima solo localmente e quindi non riusciamo a dire se lo è anche globalmente o più genericamente quanto si discosta dall'ottimo globale. Se siamo fortunati, un algoritmo di ricerca locale può anche trovare una soluzione ottima globalmente oppure una soluzione il cui valore non si discosta molto dalla soluzione ottima globale. Comunque, ci sono molti casi in cui gli algoritmi di ricerca locale si sono dimostrati una valida alternativa.

Un ruolo fondamentale è svolto dalla relazione di vicinanza: tale relazione è definita arbitrariamente e può essere sia molto restrittiva, limitando la grandezza del vicinato e quindi rendendo l'algoritmo più efficiente, sia molto permissiva, aumentando la grandezza del vicinato e quindi rendendo l'algoritmo meno efficiente. Non dovrebbe sorprendere che più è restrittiva la relazione di vicinanza e meno ci si può aspettare riguardo la bontà della soluzione: cercare una soluzione migliore su insiemi più grandi implica una probabilità maggiore di trovare un ottimo locale migliore.

Procediamo con un esempio, e, a tal fine, riprendiamo il problema VertexCover: Dato un grafo $G = (V, E)$ un ricoprimeno è un sottoinsieme $S \subseteq V$ tale che ogni arco di E ha almeno un vertice in S; vogliamo trovare un ricoprimento che abbia il minimo numero di nodi. Per questo problema lo spazio $\mathcal{S}$ delle (potenziali) soluzioni è dato da tutti i sottoinsiemi $S \subseteq V$ che formano un ricoprimento. Osserviamo che poichè a priori non sappiamo se un particolare insieme è o meno un insieme ricoprente, di fatto dovremo esaminare tutti i possibili insiemi di vertici.

Il costo di una soluzione S è la cardinalità della soluzione stessa: $costo(S) = |S|$. Dobbiamo adesso definire una relazione di "vicinanza" per poter esplorare $\mathcal{S}$ passando da una soluzione ad un'altra ad essa vicina. Una possibile relazione è la seguente: due soluzioni sono vicine se una può essere ottenuta dall'altra aggiungendo o rimuovendo un solo vertice; quindi data una soluzione S tutte le soluzioni ad essa vicine sono quelle che possiamo ottenere o rimuovendo un vertice di S, o aggiungendo un vertice di $V - S$ ad S. Dunque, per ogni S ci sono esattamente n insiemi di vertici vicini ad S, e quindi al massimo n insiemi che effettivamente sono dei ricoprimenti.

Pertanto data una soluzione S, in tempo proporzionale ad n riusciamo a considerare tutte le soluzioni vicine ad S (a tale tempo dovremo aggiungere quello necessario per controllare se un insieme è ricoprente o meno). Dunque l'algoritmo di ricerca locale per VertexCover procede nel seguente modo: data una soluzione S, con costo $|S|$, considera tutti le soluzioni vicine ad S e se ne trova una S' con costo minore di S si "sposta" nella nuova soluzione S'. Si noti che V è sicuramente un insieme ricoprente, quindi è un buon punto di partenza.

Algorithm 7: VertexCoverLS

$S \leftarrow V$
while *Esiste* S' *vicino di* S *con* $|S'| < |S|$ **do**
  $S \leftarrow S'$.
return S

L'algoritmo VertexCoverLS procede passando da una soluzione ad una con costo minore, finchè tale passaggio è possibile. Abbiamo già detto, però, che la soluzione

restituita dall'algoritmo è una soluzione ottima *localmente*, ma potrebbe non essere ottima *globalmente*. Questo è dovuto al modo in cui lo spazio delle soluzioni è stato esplorato. Possiamo fare un analogo con la ricerca del minimo di una funzione continua di una variabile in cui il valore della soluzione x è $y = f(x)$ e le soluzioni vicine ad x sono quelle in un intorno $[x - \epsilon, x + \epsilon]$ di x. Se la funzione è crescente l'algoritmo si sposterà nell'intorno sinistro, se invece è decrescente si sposterà nell'intorno destro. In un punto di minimo (locale) l'algoritmo si ferma e restituisce il punto di minimo locale, che potrebbe non essere il minimo della funzione.

Le Figure 2.18 e 2.19 mostrano dei casi dove l'algoritmo di ricerca locale si ferma in un punto di minimo locale; è possibile, come nel caso della Figura 2.20 che l'algoritmo di ricerca locale trovi il minimo globale, quando il minimo locale trovato corrisponde al minimo globale.

Nel caso di una funzione di una variabile è facile visualizzare il "percorso" seguito dall'algoritmo, come è stato mostrato nelle Figure 2.18-2.20. Per problemi reali il percorso seguito è più difficile da visualizzare ma il concetto di base è lo stesso. Per il problema VertexCover, ad esempio, dovremmo visualizzare tutti gli insiemi ricoprenti e quindi tracciare il percorso fatto. Ovviamente quali sono esattamente gli insiemi ricoprenti dipende dal grafo di partenza. Consideriamo un caso semplice in cui il grafo non ha archi cioè $E = \{\}$. In questo caso un qualunque sottoinsieme di vertici è ricoprente, incluso l'insieme vuoto, e l'algoritmo VertexCoverLS, partendo da $S = V$, rimuoverà un vertice in ogni iterazione e, dopo n iterazioni, restituirà come soluzione l'insieme vuoto che è la soluzione ottima (globale).

Se invece il grafo G è un grafo a stella come mostrato nella Figura 2.21, allora il ricoprimento ottimo è l'insieme che contiene il solo nodo centrale v_1 della stella. In questo caso l'algoritmo potrebbe non restituire tale insieme ma, invece, restituire il ricoprimento $\{v_2, \ldots, v_n\}$. Questo succede quando v_1 viene scelto nella prima iterazione come nodo da cancellare: a quel punto l'algoritmo non può cancellare nessun altro nodo in quanto non avrebbe più un insieme ricoprente.

Un altro caso sufficientemente semplice da analizzare è quello di un grafo in cui i nodi sono uniti in un unico cammino, cioè gli archi sono $E = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$. Un insieme minimo ricoprente ottimale è l'insieme che contiene i nodi con indice pari $\{v_2, v_4, \ldots\}$. In questo caso però l'algortitmo VertexCoverLS potrebbe restituire vari minimi locali. Ad esempio partendo dall'insieme ricoprente $\{v_2, v_3, v_5, v_6, v_8, v_9, \ldots\}$, cioè dall'insieme di vertici meno i vertici con indice $1 + 3k$, $k = 0, 1, \ldots$, non possiamo togliere nessun nodo, quindi questo insieme ricoprente è un minimo locale; tuttavia ha molti più nodi rispetto all'insieme ricoprente ottimale.

Un miglioramento che si può apportare alla strategia generale per dimunuire la probabilità di bloccarsi in un minimo locale è data dalla strategia detta di *simulated annealing*[5]. Un algoritmo di simulated annealing è una ricerca locale nella quale si da la possibilità, con una certa probabilità, di passare anche ad una soluzione peggiore, nella speranza che da questa soluzione peggiore poi si possa andare verso soluzioni migliori. Nel descrivere lo pseudocodice assumeremo di avere a che fare con un problema di minimizzazione (per problemi di massimizzazione basterà invertire la logica di confronto delle soluzioni).

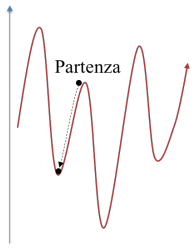

Figura 2.18: La ricerca termina in un minimo locale.

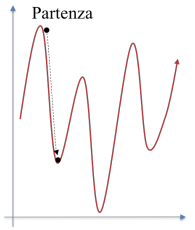

Figura 2.19: La ricerca termina in un minimo locale.

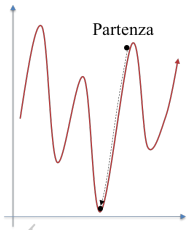

Figura 2.20: La ricerca termina in un minimo globale.

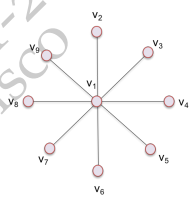

Figura 2.21: Un grafo a stella.

[5] Il nome, ricottura simulata, deriva dal fatto che l'idea si base sul processo metallurgico di fusione in cui i metalli vengono raffreddati in maniera graduale.

Algorithm 8: SIMULATEDANNEALING

```
Sia s una soluzione qualsiasi
repeat
    Scegli in modo casuale una soluzione s' vicino a s
    Δ ← costo(s') − costo(s)
    if Δ < 0 then
        s ← s'
    else
        s ← s' con probabilità e^{−Δ/T}
until s non viene cambiata
return s
```

Il parametro T, che per analogia con il processo metallurgico al quale è ispirata la strategia viene detto *temperatura*, svolge un ruolo fondamentale. Se $T = 0$ allora la probabilità di sostituire s con una soluzione peggiore s' è 0 e quindi l'algoritmo si riduce alla strategia generale che permette di spostarsi solo in soluzioni migliori. Se T invece è grande, allora la probabilità di passare a soluzioni peggiori aumenta (per $T \to \infty$ la probabilità tende a 1).

Osserviamo che la probabilità di passare a soluzioni peggiori serve per poter "sfuggire" a minimi locali. Tuttavia può provocare comportamenti poco efficienti. Si consideri di nuovo il caso del problema VERTEXCOVER su un grafo senza archi. In questo caso abbiamo osservato come la strategia standard ad ogni passo tolga un nodo per arrivare facilmente alla soluzione ottima. Ammettendo la possibilità di saltare a soluzioni peggiori, magari con alta probabilità, si ha che l'algoritmo oscilli avvicinandosi e allontandosi dalla soluzione ottima senza mai raggiungerla e con la concreta possibilità di non terminare mai.

Quindi se da un lato vogliamo una probabilità alta per poter sfuggire i punti di minimo locale, dall'altro vogliamo una probabilità bassa per evitare comportamenti poco efficienti.

Dunque, quale valore della temperatura dovremmo usare?

Il trucco della strategia di simulated annealing è quello di utilizzare un valore variabile per la temperatura, iniziando la ricerca con una temperatura alta e gradualmente riducendola a zero. Quindi inzialmente la ricerca procede in modo abbastanza libero, cioè permettendo ababstanza spesso spostamenti verso soluzioni peggiori. Poi gradualmente la temperatura viene diminuita e questo permette sempre meno di spostarsi verso soluzioni peggiori, fino ad arrivare a zero, per garantire la convergenza dell'algoritmo. La strategia (*annealing schedule*, che potremmo tradurre con *schema di raffreddamento*) con cui la temperatura viene abbassata è parte dell'algoritmo e può essere progettata in funzione dello specifico problema.

2.10 Note bibliografiche

Ulteriori approfondimenti sulla ricerca esaustiva intelligente possono essere trovati nel Capitolo 9 del libro DPV2008 [9]. Gli algoritmi RECURSIVEVC, GREEDYIS e DYNPROWIS, sono presentati nel Capitolo 10 di KT2014 [20], dove è possibile trovare ulteriori appro-

fondimenti. Per approfondimenti sugli algoritmi di ricerca locale si veda il Capitolo 4 di MP2017 [28], e il Capitolo 9 di DPV2008 [9] e il Capitolo 12 di KT2014 [20].

2.11 *Esercizi*

1. Nell'esercizio di programmazione per la scoperta dei fattori di un intero n abbiamo usato l'accorgimento di provare a dividere n per tutti gli interi fino a $n/2$ sfruttando il fatto che un intero più grande di $n/2$ non può dividere n. Analizza il vantaggio derivante da tale accorgimento. Di quanti bit bisogna "allungare" n per perdere tale vantaggio?

2. Disegnare l'albero di backtracking per la formula boolena $\phi = (\neg a + b + \neg c + d) \cdot (\neg a + \neg b) \cdot (\neg a + b) \cdot (a + \neg c) \cdot (a + c)$.

3. Si consideri l'istanza del problema LatinSquare riportata in figura 2.22 (quindi $n = 2$). Si applichi la riduzione al problema del Sudoku mostrando la corrispondente istanza di tale problema che deriva dalla riduzione.

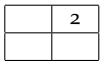

Figura 2.22: Istanza di LatinSquare per l'esercizio 3.

4. Si fornisca una restrizione sull'input, coinvolgendo almeno n elementi della scacchiera, che renda il problema LatinSquare risolvibile in tempo polinomiale. Si dia un algoritmo efficiente per la restirizione individuata.

5. Si consideri il grafo riportato in figura 2.23 che rappresenta un'istanza del problema del commesso viaggiatore. Si applichi (disegnando l'albero) la tecnica di branch-and-bound per risolvere il problema.

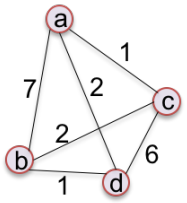

Figura 2.23: Istanza di TSP per l'esercizio 5.

6. Fornire un algoritmo di baktrack o di branch-and-bound per il problema IndependentSet: descrivere l'albero di branch-and-bound specificando in maniera dettagliata cosa rappresentano i nodi ed i cammini.

7. Consideriamo il problema 3Sat-3: Data una formula $\phi(x_1, x_2, \ldots, x_n) = C_1 \cdot \ldots \cdot C_m$ definita come la congiunzione di clausole in cui ogni clausola ha esattamente 3 letterali (com per 3Sat) e ognuna delle n variabili appare, senza o con negazione, in esattamente 3 clausole. 3Sat-3 è come 3Sat con un vincolo aggiuntivo. Mentre una formula per 3Sat potrebbe non essere soddisfattibile, una formula per 3Sat-3 è sempre soddsifattibile; quindi il problema consiste nel trovare un assegnamento che la renda vera. Sfruttare i vincoli del problema per fornire un algoritmo polinomiale per 3Sat-3.

8. Consideriamo il seguente problema di schedulazione: ci sono due macchine identiche M_1 e M_2 che devono eseguire un insieme di n lavori che richiedono tempi di esecuzione $t_1, \ldots, t_n$. I lavori devono essere partizionati in due sottoinsiemi A e B, che verranno eseguiti dalle due macchine, in modo tale che il tempo di esecuzione più grande sia minimizzato; cioè detti $T_1 = \sum_{j \in A} t_j$ e $T_2 = \sum_{j \in B} t_j$ scegliere A e B in modo tale da minimizzare $\Delta = |T_1 - T_2|$. Questo problema di scheduling è $\mathcal{NP}$-completi. Valutiamo quindi un algoritmo di ricerca locale. Come relazione di vicinanza fra le soluzioni possiamo considerate la seguente: due soluzioni sono

vicine se possiamo passare dall'una all'altra spostando un lavoro da una macchina all'altra. Una soluzione di partenza potrebbe essere quella che mette i primi $n/2$ lavori in A ed i restanti in B.

(a) Quanto è buona la soluzione che otterremo con questo algortimo di ricerca locale? Assumendo che non ci siano lavori che che dominano il tempo di esecuzione, cioè assumendo che $t_j \leq \frac{1}{2}\sum_{i=1}^{n} t_i$ per tutti i lavori $j = 1 = 2, \ldots, n$, provare che tutte le soluzioni ottime localmente si ha che $\frac{1}{2}T_1 \leq T_2 \leq 2T_1$, cioè che i tempi T_1 e T_2 sono abbastanza bilanciati.

(b) Quante volte un lavoro può essere spostato da una macchina all'altra? Cioè l'algoritmo converge verso una soluzione? Per essere sicuri che ciò accada consideriamo la seguente variante: se ci sono più lavori che possono essere spostati, allora spostiamo sempre il lavoro con il tempo di esecuzione più grande. Provare che con questa variante ogni lavoro viene spostato al massimo una volta e quindi la sequenza di ricerca della soluzione non può contenere più di n soluzioni.

(c) Fornire un esempio in cui l'algoritmo trova un minimo locale che non è globale.